Assignment 2 report

Background

Tessella is a two-player abstract board game with a unique board layout, composed of octagons and squares tessellated together, which endows the game with distinct properties for each type of tile. Each player starts with 7 pieces, placed at opposite corners of the board. Players take turns moving their pieces to adjacent empty spaces, with movement direction determined by the shape of the tile the piece occupies. Pieces on octagons can move in eight directions, while those on squares can only move straight. When two of a player's pieces are aligned horizontally, vertically, or diagonally with no other pieces in between, they gain the ability to capture an opponent's piece by moving any number of spaces along that line, without jumping over other pieces. The game is won by the first player to capture four of their opponent's pieces.

Overview

The code implements a decision-making system for a board game AI, featuring several core functions: `get_moves_for_one_pice` calculates all possible moves for a single piece, `evaluate_move` assesses the score of a move based on various factors, `choose_best_move` selects the highest-scoring move, and `play_games` serves as the main function that reads the board state, calls the decision-making functions, and saves the result. Additionally, it includes auxiliary functions such as `total_euclidean_distance` to calculate the total Euclidean distance between all pieces, `can_capture_by_firing` to determine if a piece can capture an opponent's piece via a specific path, `count_threats` to count the number of threats a piece faces, and `is_under_threat` to check if a piece is within the opponent's attack range.

get_moves_for_one_pice function

The function `get_moves_for_one_pice` calculates possible moves for a piece on a game board. It starts by unpacking the piece's position into `(x, y)` coordinates and initializing an empty list `moves` to store valid moves. It checks the piece's shape and role—if the shape isn't 'O' or 'S', or the role isn't 1 or 2, it returns the empty list.

The function defines two direction lists: `directions` (eight possible moves) and `move_dirs` (a subset for 'S' pieces). It iterates over `move_dirs`, calculates new positions `(nx, ny)` by adding direction offsets, and checks if these positions are within the board bounds and empty. If so, the move is added to `moves`.

For more complex "jumping" moves, it iterates over `directions`, calculates "backward" positions `(bx, by)` by subtracting offsets, and checks if these positions are occupied by the player's own pieces. If a valid backward position is found, it calculates "forward" positions `(cx, cy)` and checks if the path is clear. If clear, the move is added to `moves`.

Finally, the function returns the `moves` list, containing all possible moves for the piece based on the board state and positions of both players' pieces.

```
def get_moves_for_one_pice(board, pice, my_positions, enemy_positions):
    x, y = pice
    moves = []
    src_shape, src_role = board[x][y]
    if src_shape not in ('O', 'S') or src_role not in (1, 2):
        return moves

    directions = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (1, -1), (-1, 1),
(-1, -1)]
    move_dirs = directions if src_shape == 'O' else directions[:4]

    for dx, dy in move_dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < MAX_M and 0 <= ny < MAX_N and board[nx][ny][1] == 0:
            moves.append(((x, y), (nx, ny)))

    for dx, dy in directions:
        bx, by = x - dx, y - dy
        while 0 <= bx < MAX_M and 0 <= by < MAX_N:
            if (bx, by) not in my_positions:
                if board[bx][by][1] != 0: break
                bx -= dx
                by -= dy
                continue
            cx, cy = x + dx, y + dy
            while 0 <= cx < MAX_M and 0 <= cy < MAX_N:
                if (cx, cy) in my_positions: break
                if (cx, cy) in enemy positions:

                    clear = True
                    checkx, checky = bx + dx, by + dy
                    while (checkx, checky) != (cx, cy):
                        if (checkx, checky) == (x, y):
                            checkx += dx
                            checky += dy
                            continue
                        if not (0 <= checkx < MAX_M and 0 <= checky < MAX_N) or
board[checkx][checky][1] != 0:
                            clear = False
                            break
                        checkx += dx
                        checky += dy
                    if clear:
                        moves.append(((x, y), (cx, cy)))
                    break
                elif board[cx][cy][1] != 0:
                    break
                cx += dx
                cy += dy
            break
    return moves
```

evaluate_move function

The function evaluate_move assesses the quality of a potential move in a game by calculating a score based on various strategic factors. First, it identifies the current positions of all pieces with the same role and updates these positions to reflect the move, removing the piece from its old position and placing it in the new one. It also updates the enemy positions to remove any piece that might have been captured by this move.

It assesses the strategic value of a move by considering several key factors. First, it significantly increases the score if the move captures an enemy piece, especially when the opponent has few pieces remaining, and adjusts the score based on whether the captured piece is under threat in the new board state. It also applies a penalty if the new position of the piece

is under threat from the opponent, with a reduced penalty for the second player to account for the disadvantage of moving second. The function rewards moves that create opportunities to threaten the opponent's pieces by counting the number of threats and adding points accordingly. Additionally, it provides a fixed score bonus to the second player as compensation. The function encourages clustering of pieces by increasing the score if the move reduces the total Euclidean distance between the player's pieces. It also significantly boosts the score if the move protects a piece that was previously under threat, particularly when the player has few pieces left. Finally, the function increases the score if the move brings the piece closer to the center of the board (position (4,4)).

Finally, the function returns the accumulated score, which reflects the strategic value of the move based on the current board state and the positions of both players' pieces.

```
def evaluate_move(board, move, enemy_positions, player_id):
    (x1, y1), (x2, y2) = move  # Unpack the move into start and end positions
    role = board[x1][y1][1]  # Get the role of the piece being moved
    score = 0  # Initialize the score

    # Get current positions of all pieces with the same role
    old_positions = [(i, j) for i in range(MAX_M) for j in range(MAX_N) if
board[i][j][1] == role]
    my_positions = [(i, j) for i in range(MAX_M) for j in range(MAX_N) if
board[i][j][1] == role and (i, j) != (x1, y1)]
    my_positions.append((x2, y2))  # Update positions to reflect the move
    enemy_positions_after = [pos for pos in enemy_positions if pos != (x2, y2)]
# Update enemy positions

    # Create a new board state to simulate the move
    new_board = [[cell.copy() for cell in row] for row in board]
    new_board[x2][y2][1] = role  # Place the piece in the new position
    new_board[x1][y1][1] = 0  # Clear the old position

    # Scoring for capturing an enemy piece
    if (x2, y2) in enemy_positions:
        if len(enemy_positions) <= 4:
            score += 10000  # High score for capturing when the opponent has
few pieces
        else:

            if not is_under_threat(new_board, (x2, y2), role ^ 3):
                score += 3000  # Score for capturing a safe piece
            else:
                score += 2000  # Lower score if the captured piece is under
threat

    # Penalty for moving into a threatened position
    if is_under_threat(new_board, (x2, y2), role ^ 3):
        danger_penalty = 750 * (7 - len(my_positions))  # Penalty based on the
number of own pieces
        if player_id == 2:
            danger_penalty *= 0.7  # Reduce penalty for the second player
        score -= danger_penalty

    # Scoring for creating threats
    if not is_under_threat(new_board, (x2, y2), role ^ 3):
        threats, attacked_important = count_threats(new_board, (x2, y2),
my_positions, enemy_positions_after, role)
        score += threats * 800  # Score based on the number of threats created

    # Compensation for the second player
    if player_id == 2:
        score += 500  # Fixed bonus for the second player
```

```
            if not is_under_threat(new_board, (x2, y2), role ^ 3):
                score += 3000  # Score for capturing a safe piece
            else:
                score += 2000  # Lower score if the captured piece is under
threat

    # Penalty for moving into a threatened position
    if is_under_threat(new_board, (x2, y2), role ^ 3):
        danger_penalty = 750 * (7 - len(my_positions))  # Penalty based on the
number of own pieces
        if player_id == 2:
            danger_penalty *= 0.7  # Reduce penalty for the second player
        score -= danger_penalty

    # Scoring for creating threats
    if not is_under_threat(new_board, (x2, y2), role ^ 3):
        threats, attacked_important = count_threats(new_board, (x2, y2),
my_positions, enemy_positions_after, role)
        score += threats * 800  # Score based on the number of threats created

    # Compensation for the second player
    if player_id == 2:
        score += 500  # Fixed bonus for the second player
```

total_euclidean_distance function

The function total_euclidean_distance calculates the total Euclidean distance between all pairs of positions in a given list. This is useful for understanding how spread out or clustered a set of points (such as game pieces on a board) are.

```
def total_euclidean_distance(positions):
    total = 0.0  # Initialize the total distance to zero
    # Iterate over all unique pairs of positions
    for (x1, y1), (x2, y2) in itertools.combinations(positions, 2):
        # Calculate the Euclidean distance between the two positions
        dist = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
        total += dist  # Add the distance to the total
    return total  # Return the accumulated total distance
```

can_capture_by_firing function

The function can_capture_by_firing determines whether a piece can capture an enemy piece by "firing" through another piece. First, it checks if the three pieces are collinear by comparing the cross product of the vectors formed by the positions of the pieces. If the cross product is not zero, the pieces are not collinear, and the function returns False. Next, it ensures that piece B lies between pieces A and C on the line segment. This is done by checking if the coordinates of B are between the coordinates of A and C.

The function then verifies the roles of the pieces at positions A, B, and C. If any of these pieces do not match the expected roles (i.e., A and B should be the player's pieces, and C should be an enemy piece), the function returns False.

After confirming the roles, it calculates the step direction vector from A to C and normalizes it to ensure it represents single steps. It then checks the path from A to C to ensure there are no obstacles between the pieces, except for B. If any cell on this path is occupied by another piece (other than B), the function returns False.

Finally, if all checks pass, the function returns True, indicating that the piece at A can capture

the piece at C by firing through B.

```
def can_capture_by_firing(B, A, C, board, my_role, enemy_role):
    ax, ay = A  # Unpack the coordinates of piece A
    bx, by = B  # Unpack the coordinates of piece B
    cx, cy = C  # Unpack the coordinates of piece C

    # Check if the three pieces are collinear (vectors are parallel)
    v1x, v1y = ax - bx, ay - by  # Vector from B to A
    v2x, v2y = cx - bx, cy - by  # Vector from B to C
    if v1x * v2y != v1y * v2x:  # Cross product should be zero for collinear
points
        return False

    # Check if B is between A and C on the line segment
    def between(a, b, c):  # Helper function to check if b is between a and c
        return min(a, c) < b < max(a, c)

    if not (between(ax, bx, cx) or between(ay, by, cy)):  # Check if B is
between A and C
        return False

    # Verify the roles of the pieces at positions A, B, and C
    if board[ax][ay][1] != my_role:  # Check if piece at A is the player's
piece
        return False
    if board[bx][by][1] != my_role:  # Check if piece at B is the player's
piece
        return False
    if board[cx][cy][1] != enemy_role:  # Check if piece at C is the enemy's
       return False

   # Calculate the step direction vector from A to C
   dx = cx - ax  # Difference in x-coordinates
   dy = cy - ay  # Difference in y-coordinates
   steps = max(abs(dx), abs(dy))  # Number of steps to move from A to C
   dx = dx // steps  # Normalize dx to represent a single step
   dy = dy // steps  # Normalize dy to represent a single step

   # Check the path from A to C for obstacles (excluding A, B, and C)
   px, py = ax + dx, ay + dy  # Start checking from the first step after A
   while (px, py) != C:  # Continue until reaching C
       # Check if the current position is out of bounds
       if not (0 <= px < MAX_M and 0 <= py < MAX_N):
           return False
       # Check if the current position is occupied by a piece (other than B)
       if (px, py) != B and board[px][py][1] != 0:
           return False
       px += dx  # Move to the next position in the path
       py += dy

   return True  # If all checks pass, the capture is possible
```

count_threats function

The function count_threats calculates the number of threats a given position (pos) faces from enemy pieces on the board. It also identifies how many of these threats involve important pieces (referred to as attacked_important). It iterates over all pairs of the player's pieces (A) and enemy pieces (C). For each pair, it checks if the enemy piece (C) can capture the piece at pos by "firing" through the player's piece (A). This is done using the can_capture_by_firing function. If such a capture is possible, it increases the threats counter.

Additionally, the function checks for more complex scenarios involving three enemy pieces (E, C, and D). It first ensures that these pieces are collinear and that C is not between E and D. If these conditions are met, it increments the attacked_important counter, indicating that the threat involves important pieces.The function returns a tuple containing the total number of threats (threats) and the number of threats involving important pieces (attacked_important).

```
def count_threats(board, pos, my_positions, enemy_positions, my_role):
    threats = 0  # Initialize the counter for total threats
    attacked_important = 0  # Initialize the counter for threats involving
important pieces

    # Iterate over all pairs of the player's own pieces and enemy pieces
    for A in my_positions:
        if A == pos:  # Skip if A is the same as the position being evaluated
            continue
        for C in enemy_positions:
            if can_capture_by_firing(pos, A, C, board, my_role, board[C[0]]
[C[1]][1]):
                threats += 1  # Increment the threats counter if capture is
possible

            # Check for more complex scenarios involving three enemy pieces
            for E in enemy_positions:
                if E == C:  # Skip if E is the same as C
                    continue
                for D in enemy_positions:
                    if D == C or D == E:  # Skip if D is the same as C or E
                        continue

                    # Check if E, C, and D are collinear
                    v1x, v1y = E[0] - C[0], E[1] - C[1]
                    v2x, v2y = D[0] - C[0], D[1] - C[1]
                    if v1x * v2y != v1y * v2x:  # Skip if not collinear
                        continue

                    # Check if C is between E and D (but not in the middle)
                    def between(a, b, c):
                        return min(a, c) < b < max(a, c)

                    if between(E[0], C[0], D[0]) or between(E[1], C[1], D[1]):
                        continue

                    # If all conditions are met, increment the
 attacked_important counter
                    attacked_important += 1

    return threats, attacked_important  # Return the total threats and threats
 involving important pieces
```

is_under_threat function

The function is_under_threat determines whether a given position (pos) on the board is under threat of being captured by the opponent. First, it unpacks the coordinates of the position being evaluated and retrieves the role of the piece at that position (my_role). It then collects the positions of all enemy pieces on the board by iterating over the entire board and identifying cells where the role matches enemy_role.

The function then iterates over all pairs of enemy pieces (B and A). For each pair, it checks if the piece at position B can capture the piece at pos by "firing" through the piece at position A. This is done using the can_capture_by_firing function. If any such capture is possible, the function immediately returns True, indicating that the position is under threat.

If no such capture is found after checking all pairs of enemy pieces, the function returns False, indicating that the position is not under threat.

```
def is_under_threat(board, pos, enemy_role):
    x, y = pos  # Unpack the coordinates of the position being evaluated
    my_role = board[x][y][1]  # Retrieve the role of the piece at the given
position

    # Collect the positions of all enemy pieces on the board
    enemy_positions = [
        (i, j) for i in range(MAX_M) for j in range(MAX_N)
        if board[i][j][1] == enemy_role
    ]

    # Iterate over all pairs of enemy pieces
    for B in enemy_positions:  # B is the potential "firing" piece
        for A in enemy_positions:  # A is the potential "assisting" piece
            if A == B:  # Skip if A and B are the same piece
                continue
            # Check if the piece at B can capture the piece at pos by firing
through A
            if can_capture_by_firing(B, A, pos, board, enemy_role, my_role):
                return True  # If capture is possible, the position is under
threat

    return False  # If no capture is possible, the position is not under threat
```

choose_best_move function

The function choose_best_move is designed to select the optimal move for a player based on the current state of the game board. It initializes variables to track the best move and its corresponding score. It starts by setting best_move to None and best_score to negative infinity. It also initializes an empty list all_moves to store all possible moves for the player's pieces.

The function then iterates over each piece in my_positions and generates all possible moves for that piece using the get_moves_for_one_pice function. Each move is added to the all_moves list. For each move, it calculates a score using the evaluate_move function, which assesses the strategic value of the move based on various factors such as capturing enemy pieces, creating threats, and protecting own pieces.

If the score of a move is higher than the current best_score, the function updates best_score and sets best_move to the current move. After evaluating all possible moves, the function returns the best move and its corresponding score.

```
def choose_best_move(board, my_positions, enemy_positions, player_id):
    best_move = None  # Initialize the best move to None
    best_score = float('-inf')  # Initialize the best score to negative infinity
    all_moves = []  # Initialize an empty list to store all possible moves

    # Iterate over each piece in the player's positions
    for pice in my_positions:
        # Get all possible moves for the current piece
        moves = get_moves_for_one_pice(board, pice, my_positions, enemy_positions)
        all_moves += moves  # Add the moves to the list of all possible moves

        # Evaluate each move
        for move in moves:
            # Calculate the score for the current move
            score = evaluate_move(board, move, enemy_positions, player_id)
            # Update the best move and score if the current move has a higher
score
            if score > best_score:
                best_score = score
                best_move = move

    return best_move, best_score  # Return the best move and its score
```

play_games function

First, the function reads the current state of the game, including the positions of both players' pieces and the game board, using the read_ckbd function. It determines the current player (player_id) based on whether the turn number is odd or even. The positions of the current player's pieces (my_positions) and the opponent's pieces (enemy_positions) are then assigned accordingly.

Next, the function calls choose_best_move to evaluate all possible moves and select the best one based on a scoring system. The best move and its corresponding score are stored in best_move and best_score, respectively.

The function then unpacks the best move into the starting and ending positions (pice_from and pice_to). It prints the score of the best move and the move itself to provide feedback on the decision. Finally, it saves the decision using the save_decision function, which likely updates the game state or records the move for future reference.

```
def play_games(step: int) -> None:
    # Read the current state of the game (positions and board) from the previous
step
    playerA_positions, playerB_positions, board = read_ckbd(step - 1)

    # Determine the current player based on the step number
    player_id = 1 if step % 2 == 1 else 2

    # Assign positions based on the current player
    my_positions = playerA_positions if player_id == 1 else playerB_positions
    enemy_positions = playerB_positions if player_id == 1 else playerA_positions

    # Choose the best move using the choose_best_move function
    best_move, best_score = choose_best_move(board, my_positions, enemy_positions,
player_id)

    # Unpack the best move into starting and ending positions
    pice_from, pice_to = best_move

    # Print the score and details of the best move
    print("show my score:", best_score)
    print("show my move:", pice_from, pice_to)

    # Save the decision for the game to proceed
    save_decision(pice_from[0], pice_from[1], pice_to[0], pice_to[1])
```

Simulate the complete process

Each round starts by determining the current state of the game based on the given step. The play_games function reads the positions of both players' pieces and the game board using read_ckbd. It then identifies the current player based on whether the step number is odd or even. The positions of the current player's pieces (my_positions) and the opponent's pieces (enemy_positions) are assigned accordingly. Next, the function choose_best_move is called to evaluate all possible moves for the current player's pieces. For each piece, get_moves_for_one_pice generates all possible moves, and each move is scored using evaluate_move, which considers factors such as capturing enemy pieces, creating threats, protecting own pieces, and moving towards the center of the board. The move with the highest score is selected as the best move. The best move and its score are printed, and the

decision is saved using save_decision, which updates the game state or records the move for future reference. This process effectively simulates a turn in the game, ensuring that the current player makes an optimal move based on the current board state.

Strengths

Comprehensive Move Evaluation: The AI evaluates moves based on multiple strategic factors, including capturing enemy pieces, creating threats, protecting own pieces, clustering pieces together, and moving towards the center of the board. This multi-faceted evaluation helps in making well-rounded decisions that consider both offensive and defensive aspects.

Adaptive Strategy: The AI adjusts its strategy based on the current state of the game. For instance, it gives higher scores to moves that capture enemy pieces when the opponent has few pieces left, and it provides a bonus for the second player to compensate for the disadvantage of moving second. This adaptability allows the AI to respond effectively to different game scenarios.

Threat Detection and Mitigation: The AI includes mechanisms to detect threats to its own pieces and to create threats for the opponent's pieces. By using functions like is_under_threat and count_threats, it can identify vulnerable positions and capitalize on opportunities to threaten the opponent's pieces, enhancing its strategic depth.

Weaknesses

Computational Complexity: The AI's move evaluation process involves multiple nested loops and checks, which can be computationally expensive. Functions like count_threats and can_capture_by_firing involve iterating over all enemy pieces and checking various conditions, leading to a high time complexity. This could result in slower decision-making as the number of pieces on the board increases, potentially affecting the AI's performance in real-time scenarios.

Limited Lookahead: The current implementation evaluates moves based on the immediate state of the board without considering future moves. It does not perform any form of lookahead or minimax search, which means it may not anticipate the opponent's responses effectively. This could lead to suboptimal moves in situations where the opponent has a strong countermove.

Reference

https://boardgamegeek.com/thread/2144227/tessella-2019-two-player-pnp-game-design-contest

All code is implemented by myself.